🔐 Module 2: Keys, Constraints, and Dependencies

Why Order Matters in a Relational World

🎯 Lesson Objective

By the end of this module, you will:

- Understand the roles of **primary keys** and **foreign keys** in relational tables
- Know how **NULL values** behave in different contexts
- Learn the basics of **functional dependencies** and **Armstrong's Axioms**
- Understand the meaning of **entity** and **instance** in table design
- See how missing or broken constraints can lead to real-world database failures

---

🧱 Why This Matters

If a database can't tell one record from another, or can't enforce relationships between its parts, your data becomes messy—and potentially dangerous. Keys and constraints are how a relational database avoids confusion and chaos. They're essential for data accuracy, searchability, and trust.

🧱 Section 1: The Foundation—Keys

Let's build on what we learned in Module 1.

📘 In a relational database, a **table** usually represents an **entity**—a real-world category of things you want to track.

📘 An **entity** is something like a person, product, course, or event.

📘 Each **row** in that table is one **instance** (or **record**) of the entity—a specific person, one course, one sale, one match.

For example:

- A **Players** table describes the *Player* entity
- The row with PlayerID = 101 is one *instance* of a player

🔑 Primary Keys: Naming Each Instance

📘 A **primary key** is a column (or set of columns) that uniquely identifies each **instance** in an entity table. Every row must have one, and no two can be the same.

Example—our tennis club's Players table:

PlayerID	Name	Age	MembershipStatus
101	Sam	32	Active
102	Jordan	29	Inactive
103	Taylor	41	Active

- This table describes the **Player** entity
- PlayerID is the **primary key**
- Each row is a **player instance** with a unique ID

❗ *Don't use names as keys!* People share names. Always use a generated ID or something guaranteed to be unique.

🔗 Foreign Keys: Connecting Entities

Often, one entity relates to another. For that, we use **foreign keys**.

📘 A **foreign key** is a column that links to the **primary key** of another table. It connects an instance in one entity to an instance in another.

Now let's track matches:

MatchID	Date	Player1ID	Player2ID
201	2025-06-10	101	102
202	2025-06-11	103	101

- MatchID is the **primary key** of the Match entity
- Player1ID and Player2ID are **foreign keys** pointing to **instances** in the Player entity

🖼 *Visual Idea:* Draw two boxes labeled "Players" and "Matches." Use arrows from PlayerID (Players table) to Player1ID and Player2ID (Matches table) to show foreign key relationships.

This lets the database:

- Confirm those players exist
- Prevent errors like scheduling matches for nonexistent players
- Let you query across entities ("Show all matches played by Jordan")

📌 Common Constraint Types

Constraint Type	What It Does
PRIMARY KEY	Ensures each row is unique and not NULL
FOREIGN KEY	Ensures references to another table are valid
NOT NULL	Column must have a value
UNIQUE	No duplicate values in that column

❌ Section 2: NULL Values and Constraint Logic

📘 A **NULL** means the value is unknown, missing, or doesn't apply. It's *not* zero or empty—it's undefined.

PlayerID	Name	Age	Email

101	Sam	32	sam@email.com
102	Jordan	29	NULL
103	Taylor	41	taylor@email.com

This matters because NULL behaves differently in queries.
SELECT * FROM Players WHERE Email != 'sam@email.com';
Jordan is **not returned**—because NULL isn't "not equal," it's "unknown."

📘 **What is SQL?**

SQL (Structured Query Language) is the standard language used to interact with relational databases. It lets you ask questions, retrieve data, insert new records, and enforce rules. While you won't learn SQL deeply in this course, you'll occasionally see small examples to show how the database behaves.

Primary keys can never be NULL (every instance must be identifiable), but **foreign keys** *can* be NULL—if a relationship doesn't yet exist.

🧠 **Section 3: Functional Dependencies**

📘 A **functional dependency** means one column (or group of columns) determines the value of another column.

Analogy: Think of a **library card number**: If you know the card number, you can find the exact person it belongs to—but not the other way around.

Card Number → Person

Example:
PlayerID → Name
This means: once you know the PlayerID, you can determine the Name—because each instance has exactly one name.
More complete:
PlayerID → Name, Age, Email
This concept helps us:

- Detect when columns belong together
- Spot redundancy
- Normalize our design (coming in Module 3)

📀 **Section 4: Armstrong's Axioms — Visualized with Tennis Club Data**

📘 *Armstrong's Axioms* are a small set of logical rules used to infer all the functional dependencies that must hold in a table.
They help you reason through whether your table structure makes sense—or is bound to fail.
Let's break them down with a clear, real-world scenario from your tennis club:

🎾 **Sample Table: Match Results**

MatchID	Court	Date	WinnerID	WinnerName
201	CourtA	2025-06-10	101	Sam
202	CourtB	2025-06-11	103	Taylor

This table describes the **Match** entity. Each row is one **match instance**.

---

1. 🪞 Reflexivity

If you know a group of columns, then you also know any subset of them.

Example:

If we know both WinnerID and WinnerName, then we certainly know WinnerName.

In dependency notation:

WinnerID, WinnerName → WinnerName ✅

---

2. ➕ Augmentation

If one dependency holds, then adding the same column(s) to both sides still makes a valid rule.

Example:

If WinnerID → WinnerName,
then we can add Court to both sides:
WinnerID, Court → WinnerName, Court ✅

---

3. 🔗 Transitivity

If A → B and B → C, then A → C.

Example using the Player table:

PlayerID	Name	Email
101	Sam	sam@email.com
103	Taylor	taylor@email.com

From our Match table:

- WinnerID → WinnerName
  And from our Player table:
- WinnerName → Email
  Therefore:
- WinnerID → Email ✅

🎓 **Summary Table of Axioms and Tennis Club Examples**

Axiom	Description	Tennis Club Example
Reflexivity	Subset rule	WinnerID, WinnerName → WinnerName
Augmentation	Add same columns to both sides	WinnerID → WinnerName ⇒ WinnerID, Court → WinnerName, Court
Transitivity	Follow the logic chain across columns	WinnerID → WinnerName, WinnerName → Email ⇒ WinnerID → Email

---

💣 Section 5: Real-World Constraint Failures

Relational databases depend on keys and constraints to keep data safe. When they're ignored? Mistakes become disasters.

🤦 Horror Story #1: Ghost Orders

- Orders table had a CustomerID (foreign key), but no constraint
- Customer was deleted
- Orders stayed behind—assigned to "no one"
- Sales reports failed; refunds couldn't be issued properly

⛔ Horror Story #2: Duplicate Patients

- Hospital used Name + Birthdate as a key
- Two John Smiths born on the same day were merged
- One received lab results meant for the other

📉 Horror Story #3: Orphaned Grades

- Student table was deleted, but their course grades were not
- Transcript system broke—missing references
- School spent weeks rebuilding records by hand

Lesson: Good keys and constraints don't just keep things tidy—they protect lives, money, and reputations.

✅ Self-Check Quiz

1. What is an **entity** in a database?
2. What is the **difference between a primary key and a foreign key**?
3. True or False: A NULL value means "0"
4. Which of these is a functional dependency?
   a) MatchID → Player1ID
   b) Age → PlayerID
   c) Name → MatchID
   → *(Correct: a)*

📘 Glossary

Term	Definition
Entity	A real-world thing or concept represented by a table
Instance	One specific row in a table—one example of the entity
Primary Key	Uniquely identifies each instance in a table
Foreign Key	Connects an instance in one table to another instance elsewhere
NULL	A placeholder for unknown or inapplicable data
Functional Dependency	A rule where one column determines another
Armstrong's Axioms	Logical rules for inferring dependencies
Constraint	A rule that the database enforces to maintain integrity
SQL	Structured Query Language, used to interact with relational databases

---

💡 Real-World Design Prompt

Pick a system you use—like:

- A food delivery app
- A college enrollment system
- A local library

Ask:

- What **entity** could be a table?
- What would the **primary key** be?
- What **relationships** would need a foreign key?

🎮 *Try This:* Draw two small tables and connect them with a foreign key. You're modeling a relational system!